4. Programmation logique

4.1. Initiation au Prolog

4.1.1. Informations générales

La *programmation logique* est un paradigme de programmation basé sur la **logique des prédicats et la logique du 1er ordre**.

Il n'y a pas de fonction dans ce paradigme. L'objet central est le **prédicat**.

Dans ce paradigme, des **faits** et des **règles** sont déclarés, ce qui forme une base de connaissances. Par la suite, nous pouvons poser des questions au système par l'intermédiaire de **requêtes**.

Voici un historique succinct.

- Années 1960 : premières apparitions du paradigme de programmation logique.
- 1969 : conception du langage PLANNER par C. Hewitt, considéré comme le 1er langage de programmation logique.
- 1972 : conception du langage PROLOG par A. Colmerauer et P. Roussel.
- Fin des années 1970 à aujourd'hui : développement d'extensions du PROLOG (amélioration des performances, intégration d'une approche orientée objets, prise en compte des systèmes distribués, *etc.*).

La programmation logique a beaucoup été utilisée pour la mise au point de **systèmes experts**, de **résolveurs de problèmes** et de **traitement automatique des langues**.

En programmation logique, un **fait** est une vérité déclarée comme telle. Ce sont des vérités relatives au contexte de travail.

Exemples

Voici quelques faits.

- Le nombre 24 est pair.
- Randy est le père de Stan.
- La ligne de métro 1 s'arrête à la station Z.
- La ligne de métro 1 ne s'arrête pas à la station Z.

Certaines vérités sont plus complexes et nécessitent une abstraction par le biais de **variables universelles**. Elles peuvent être **instanciées** dans certains cas.

Exemples

Voici quelques vérités nécessitant des variables.

- La ligne de métro 1 s'arrête à toutes les stations.
  Ceci est équivalent à dire que la ligne de métro 1 s'arrête à la station x.
- Les chercheur.e.s x et y du laboratoire ont publié un article en commun.
  Cela signifie que toute paire de chercheur.e.s du laboratoire a publié un article en commun.

Une **règle** est un fait soumis à des conditions. Il est de la forme

$$\text{si } C \text{ alors } F$$

où C est une condition qui, lorsqu'elle est vérifiée, assure que F est un fait.

Exemples

Voici quelques règles.

- Si l'entier n est divisible par 4, alors n est pair.
- Si la ligne de métro x est la 2 ou la 3, alors x s'arrête à la station Z.
- Si la note d'examen n_1 de l'étudiant.e e est supérieure à 72, alors le cours est validé.
  Si la note d'examen n_1 de l'étudiant.e e est supérieure à 29 et la note de devoir n_2 de l'étudiant e supérieure à 99, alors le cours est validé.

Remarque : un **fait** est finalement une règle **si** C **alors** F dont la **condition** C est **toujours vraie**. Cela revient bien à dire que F est vraie.

Une **requête** est une question posée, relativement à un ensemble de faits et de règles. Le processus qui crée une réponse **affirmative** ou **négative** est sa **résolution**.

La résolution est affirmative lorsque le système parvient à **démontrer** la requête. Il se base sur les faits et les règles qui le constituent.

Lorsque le système ne parvient pas à démontrer que la requête est vraie, la résolution est négative. Il s'agit de la **négation par l'échec**.

Exemples

Soient les faits et règles suivants :

1. Randy est le père de Stan.
2. Randy est le père de Shelly.
3. Sharon est la mère de Stan.
4. Si x est le père de y et x est le père de z, alors y et z sont dans la même fratrie.

Voici quelques requêtes et leurs résolutions.

- Soit la requête « Stan et Shelly sont dans la même fratrie. ». Elle admet une résolution affirmative car en posant $x = \texttt{Randy}$, $y = \texttt{Stan}$ et $z = \texttt{Shelly}$, la règle 4 s'applique. Les règles 1 et 2 montrent que la prémisse de cette règle est vraie, entrainant sa conclusion.
- La requête « Sharon est la mère de Shelly. » ne peut pas être démontrée. Sa résolution est donc négative.

Une requête peut contenir des **variables**. Dans ce cas, le processus de résolution va chercher l'**ensemble de ses solutions**. Il s'agit de l'ensemble des des **spécialisations** des variables qui donnent des requêtes dont la résolution est affirmative.

Exemples

Soient les faits et règles suivants :

1. Randy est le père de Stan.
2. Randy est le père de Shelly.
3. Sharon est la mère de Stan.
4. Si x est le père de y et x est le père de z, alors y et z sont dans la même fratrie.

- La requête « a est le père de b. » admet comme ensemble de solutions

$$\{[a = \texttt{Randy}, b = \texttt{Stan}],\ [a = \texttt{Randy}, b = \texttt{Shelly}]\}.$$

- La requête « a et b sont dans la même fratrie. » admet comme ensemble de solutions

$$\{[a = \texttt{Stan}, b = \texttt{Shelly}],\ [a = \texttt{Shelly}, b = \texttt{Stan}],\ [a = \texttt{Stan}, b = \texttt{Stan}],\ [a = \texttt{Shelly}, b = \texttt{Shelly}]\}.$$

4.1.2. Programmer en Prolog

Nous utilisons l'implantation SWI-Prolog. Il en existe d'autres (qui ne seront pas considérés dans ce cours).

Voici quelques outils et références en lien :

- site du projet : https://www.swi-prolog.org/ ;
- plateforme interactive en ligne : https://swish.swi-prolog.org/ ;
- sur les systèmes Linux, l'exécutable est `swipl` ;
- voir les références sur la programmation logique données dans les 1res pages de ce cours.

Nous utiliserons `swipl` principalement mais la plateforme interactive est très utile. Elle contient en plus cela de nombreux exemples.

Le programme `swipl` travaille sur des fichiers d'extension `.pl`. Il s'agit d'un **interpréteur** qui **résout des requêtes**.

Voici quelques prédicats prédéfinis utiles :

- `consult("A.pl").` importe les déclarations du fichier `A.pl` dans la session ;
- `reconsult("A.pl").` importe à nouveau et sans les dupliquer les déclarations du fichier `A.pl` dans la session ;
- `make.` importe à nouveau tout fichier déjà importé qui a subi une modification depuis sa dernière importation ;
- `listing.` affiche toutes les règles de la session ;
- `trace.` active des informations de débogage. La commande `notrace.` permet de les désactiver ;
- `help(X).` affiche de la documentation sur l'entité `X` ;
- `halt.` quitte la session.

Il est possible de charger directement plusieurs fichiers `F1.pl`, ..., `Fn.pl` par la commande

```
swipl -s F1.pl -s F2.pl ... -s Fn.pl
```

Il est aussi possible d'ajouter des options `-g Q` pour demander la résolution de la requête `Q`.

Un *terme* est défini récursivement comme étant

- soit une *variable*, qui commence par une majuscule ou un tiret bas ;
- soit un *atome*, qui commence par une minuscule, attaché à une suite de termes placés entre parenthèses et séparés par des virgules.

L'*arité* d'un atome `a` est le nombre `n` de termes sur lequel il s'applique. Ceci est noté `a/n`.

Exemples

- L'assemblage syntaxique

  ```
  aime(Chat, croquettes(marque(minou_maxou), type(chats_sportifs)))
  ```

  est un terme.

  L'atome `aime` est d'arité 2. Son premier argument est la variable `Chat`.

  L'atome `chats_sportifs` est d'arité 0. Il ne prend aucun argument et les parenthèses sont donc omises.

- L'assemblage syntaxique

  ```
  egaux(moins(X, Y), plus(moins(moins(X)), moins(Y)))
  ```

  possède plusieurs occurrences différentes de l'atome `moins`. Certaines vérifient `moins/2` et d'autres, `moins/1`. Ce sont des atomes différents.

Une *clause* est un assemblage syntaxique de la forme

```
t :- t1, ..., tn.
```

où `t` est un terme appelé *tête* et `t1, ..., tn` est une conjonction de termes appelée *corps*.

Cette clause exprime la règle

si t_1 et ... et t_n alors t.

Pour exprimer un fait, la syntaxe devient simplement

```
t.
```

Exemples

- La clause (qui est un fait)

```
pere(randy, stan).
```

  exprime que Randy est le père de Stan.

- La clause (qui est une règle)

```
fratrie(X, Y) :- pere(Z, X), pere(Z, Y).
```

  exprime le fait que `X` et `Y` sont dans la même fratrie s'il existe un père `Z` commun.

Un *prédicat* est une **suite de clauses** dont la tête est formée du même atome.

Exemple

Les trois clauses

```
pere(randy, stan).
pere(randy, shelly).
pere(marvin, randy).
```

définissent le prédicat `pere`.

Un *programme* est une **suite de prédicats**.

Exemple

Les trois prédicats

```
pere(randy, stan).
pere(randy, shelly).
pere(marvin, randy).

fratrie(X, Y) :- pere(Z, X), pere(Z, Y).

grand_pere(X, Y) :- pere(X, Z), pere(Z, Y).
```

forment un programme.

Une *requête* est un assemblage syntaxique de la forme

```
t1, ..., tn.
```

où `t1, ..., tn` est une conjonction de termes.

Il est possible de la mentionner dans l'interpréteur ou bien via l'option `-g` comme vu précédemment.

La *résolution* d'une requête est le processus qui consiste à **démontrer qu'elle est vraie**. Plusieurs types de réponses sont possibles :

- lorsque la requête est une conséquence des prédicats du programme, c'est-à-dire que chaque terme `t` de la conjonction peut être démontré, `true.` est imprimé ;
- lorsque la requête n'est pas une conséquence des prédicats du programme, `false.` est imprimé ;
- lorsque des variables figurent dans la requête, toutes les façons de rendre la requête conséquence des prédicats du programme en spécialisant les variables sont imprimées.

Lorsqu'une requête admet **plusieurs solutions**, le symbole `;` est imprimé juste après avoir imprimé une solution. L'utilisateur peut presser **ENTRÉE** pour interrompre la recherche ou bien **ESPACE** pour **rechercher la prochaine solution** potentielle.

Si la résolution s'engage dans une voie qui **ne démontre pas** la requête, `false.` est imprimé (même si la résolution de la requête est affirmative via une solution trouvée précédemment).

Exemple

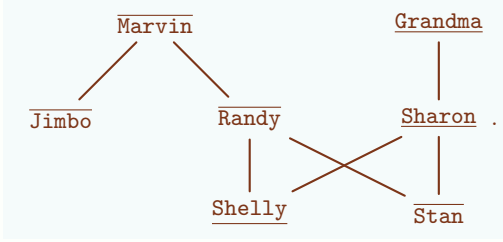
Considérons le programme

```
femelle(grandma).
femelle(sharon).
femelle(shelly).

male(marvin).
male(randy).
male(jimbo).
male(stan).

parent(marvin, randy).
parent(marvin, jimbo).
parent(grandma, sharon).
parent(randy, shelly).
parent(randy, stan).
parent(sharon, shelly).
parent(sharon, stan).
```

Ces faits modélisent l'« arbre » généalogique



Il se lit de haut en bas. Les noms de la forme NOM correspondent à des femelles et les noms de la forme $\overline{\text{NOM}}$, à des mâles.

Dans les exemples qui vont suivre, nous allons considérer des prédicats supplémentaires et observer les réponses obtenues à plusieurs requêtes.

Exemples

Voici quelques requêtes simples et leurs réponses :

- ```
  ?- male(randy). % ENTRÉE
  true.
  ```

  La réponse est `true.` car le système démontre que `male(randy)` est vrai comme conséquence qu'il s'agit d'un fait.

- ```
  ?- parent(marvin, randy). % ENTRÉE
  true.
  ```

  La réponse est `true.` car le système démontre que `parent(marvin, randy)` est vrai comme conséquence qu'il s'agit d'un fait.

- ```
  ?- parent(marvin, stan). % ENTRÉE
  false.
  ```

  La réponse est `false.` car le système ne parvient pas à démontrer que `parent(marvin, stan)` est vrai. La réponse à cette requête est donc négative.

- ```
  ?- male(stan), parent(sharon, shelly). % ENTRÉE
  true.
  ```

  La réponse est `true.` car le système démontre que les deux sous-requêtes sont vraies (il s'agit d'un « et » logique).

Exemples

Voici quelques requêtes avec des variables et leurs réponses :

- ```
  ?- male(X). % ENTRÉE
  X = marvin ; % ESPACE
  X = randy ; % ESPACE
  X = jimbo ; % ESPACE
  X = stan.
  ```

  Cette requête contient une variable `X`. Sa résolution consiste à trouver toutes les spécialisations α de `X` qui rendent `male(`α`)` affirmative.

- ```
  ?- parent(marvin, X). % ENTRÉE
  X = randy ; % ESPACE
  X = jimbo.
  ```

  Le 1er argument de `parent` est un atome et le 2e est une variable. Les spécialisations qui rendent la requête affirmative sont données.

- ```
  ?- parent(marvin, X), parent(X, stan). % ENTRÉE
  X = randy ; % ESPACE
  false.
  ```

  La même variable apparaît dans les deux sous-requêtes. Nous cherchons donc les spécialisations α de `X` telles que Marvin est parent de α et α est parent de Stan.

  Le `false` signifie que la recherche a été poursuivie sans apporter de nouvelle solution.
  ```

Exemples

Voici quelques requêtes avec des variables et leurs réponses :

- ```
  ?- parent(marvin, X), parent(Y, stan). % ENTRÉE
  X = Y, Y = randy ; % ESPACE
  X = randy, Y = sharon ; % ESPACE
  X = jimbo, Y = randy ; % ESPACE
  X = jimbo, Y = sharon.
  ```

  Cet exemple ressemble au précédent, mais une 2e variable est considérée dans la 2e requête.

  Cela donne comme solutions les spécialisations α et β de `X` et `Y` telles que Marvin est parent de α et β est parent de Stan.

- ```
  ?- parent(X, Y), parent(Y, X). % ENTRÉE
  false.
  ```

  Cette requête est **négative** car le système ne trouve **aucun exemple** de α et β tels que α est parent de β et β est parent de α.

- ```
  ?- parent(X, shelly), femelle(X). % ENTRÉE
  X = sharon.
  ```

  Nous cherchons les α tels que α est parent de Shelly et est une femelle.
  ```

Exemples

Ajoutons au programme précédent les deux prédicats

```
pere(X, Y) :-
    male(X),
    parent(X, Y).
```

Ce prédicat met `X` et `Y` en relation si `male(X)` et `parent(X, Y)`. C'est le cas lorsque `X` est un mâle et `X` est parent de `Y`.

```
mere(X, Y) :-
    femelle(X),
    parent(X, Y).
```

Ce prédicat met `X` et `Y` en relation si `X` est une femelle et `X` est parent de `Y`.

Voici quelques requêtes bâties autour de ces prédicats :

- ```
  ?- pere(X, stan). % ENTRÉE
  X = randy ; % ESPACE
  false.
  ```

  Il n'y a que Randy qui soit le père de Stan.

- ```
  ?- mere(X, Y). % ENTRÉE
  X = grandma, Y = sharon ; % ESPACE
  X = sharon, Y = stan ; % ESPACE
  X = sharon, Y = shelly ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est mère de `Y`.

Exemples

Ajoutons au programme précédent les trois prédicats

```
grand_parent(X, Y) :-
    parent(X, Z),
    parent(Z, Y).
```

```
grand_pere(X, Y) :-
    male(X),
    grand_parent(X, Y).
```

```
grand_mere(X, Y) :-
    femelle(X),
    grand_parent(X, Y).
```

Le 1er met en relation `X` et `Y` s'il existe un `Z` tel que `X` est parent de `Z` et `Z` est parent de `Y`. Les deux autres mettent en relation `X` et `Y`, si en plus de ces conditions, `X` est un mâle (pour le 2e) ou une femelle (pour le 3e).

Voici quelques requêtes bâties autour de ces prédicats :

- ```
  ?- grand_pere(X, Y). % ENTRÉE
  X = marvin, Y = stan ; % ESPACE
  X = marvin, Y = shelly ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est un grand-père de `Y`.

- ```
  ?- grand_mere(X, stan). % ENTRÉE
  X = grandma ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` telles que `X` est une grand-mère de Stan.

Exemples

Ajoutons au programme précédent les trois prédicats

```
fratrie(X, Y) :-
    parent(Z, X),
    parent(Z, Y),
    X \= Y.
```

```
frere(X, Y) :-
    male(X),
    fratrie(X, Y).
```

```
soeur(X, Y) :-
    femelle(X),
    fratrie(X, Y).
```

Le 1er met en relation `X` et `Y` s'il existe un `Z` tel que `Z` est parent de `X` et `Z` est parent de `Y` avec `X` et `Y` différents. Les deux autres mettent en relation `X` et `Y`, si en plus de ces conditions, `X` est un mâle (pour le 2e) ou une femelle (pour le 3e).

Voici quelques requêtes bâties autour de ces prédicats :

- ```
  ?- frere(X, Y). % ENTRÉE
  X = randy, Y = jimbo ; % ESPACE
  X = jimbo, Y = randy ; % ESPACE
  X = stan, Y = shelly ; % ESPACE
  X = stan, Y = shelly.
  ```

  Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est un frère de `Y`.
  Il y a une symétrie des solutions lorsque ces spécialisations portent sur deux mâles.
  La dernière solution est dupliquée.

- ```
  ?- frere(X, Y), soeur(Y, X). % ENTRÉE
  X = stan, Y = shelly ; % ESPACE
  X = stan, Y = shelly ; % ESPACE
  X = stan, Y = shelly ; % ESPACE
  X = stan, Y = shelly.
  ```

  Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est frère de `Y` et `Y` est sœur de `X`.
  Quatre solutions égales sont construites.

Exemples

Ajoutons au programme précédent le prédicat

```
oncle_ou_tante(X, Y) :-
    frere(X, Z),
    parent(Z, Y).
oncle_ou_tante(X, Y) :-
    soeur(X, Z),
    parent(Z, Y).
```

Ce prédicat met `X` et `Y` en relation s'il existe `Z` tel que `X` est frère de `Z` et `Z` est parent de `Y` ou s'il existe `Z` tel que `X` est sœur de `Z` et `Z` est parent de `Y`.

Voici quelques requêtes bâties autour de ce prédicat :

-
```
?- oncle_ou_tante(X, shelly). % ENTRÉE
X = jimbo ; % ESPACE
false.
```
Ceci donne tous les spécialisations de `X` telles que `X` est oncle ou tante de Shelly.

-
```
?- oncle_ou_tante(X, Y). % ENTRÉE
X = jimbo, Y = stan ; % ESPACE
X = jimbo, Y = shelly ; % ESPACE
false.
```
Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est oncle ou tante de `Y`.

Exemples

Ajoutons au programme précédent le prédicat

```
ancetre(X, Y) :-
    parent(X, Y).
ancetre(X, Y) :-
    parent(X, Z),
    ancetre(Z, Y).
```

Ce prédicat **récursif** met `X` et `Y` en relation si `X` est parent de `Y` ou s'il existe `Z` tel que `X` est parent de `Z` et `Z` est **récursivement en relation** avec `Y`.
Ceci revient à dire que `X` est ancêtre de `Y`.

Voici quelques requêtes bâties autour de ce prédicat :

- ```
  ?- ancetre(X, shelly). % ENTRÉE
  X = randy ; % ESPACE
  X = sharon ; % ESPACE
  X = marvin ; % ESPACE
  X = grandma ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` telles que `X` est ancêtre de Shelly.

- ```
  ?- ancetre(sharon, X). % ENTRÉE
  X = stan ; % ESPACE
  X = shelly ; % ESPACE
  false.
  ```

  Ceci, à l'inverse, donne toutes les spécialisations de `X` qui sont descendants de Sharon.

- ```
  ?- ancetre(X, Y), X = randy. % ENTRÉE
  X = randy, Y = stan ; % ESPACE
  X = randy, Y = shelly ; % ESPACE
  false.
  ```

  Cette requête est équivalente à la requête `ancetre(X, Y), randy = X.` ou encore à `ancetre(randy, Y).`.
  ```

Exemples

Ajoutons au programme précédent le prédicat

```
famille(X, Y) :-
    ancetre(X, Y).
famille(X, Y) :-
    ancetre(Y, X).
famille(X, Y) :-
    ancetre(Z, X),
    ancetre(Z, Y).
```

```
famille(X, Y) :-
    ancetre(X, Z),
    ancetre(Y, Z).
```

Ce prédicat **récursif** met `X` et `Y` en relation si `X` est ancêtre de `Y` ou si `Y` est ancêtre de `X` ou s'il existe `Z` tel que `Z` est ancêtre commun à `X` et à `Y`. Ceci revient à dire que `X` et `Y` appartiennent à la même famille.

Voici une requête bâtie autour de ce prédicat :

```
?- famille(sharon, X). % ENTRÉE
X = stan ; % ESPACE
X = shelly ; % ESPACE
X = grandma ; % ESPACE
X = sharon ; % ESPACE
X = stan ; % ESPACE
X = shelly ; % ESPACE
X = randy ; % ESPACE
```

```
X = sharon ; % ESPACE
X = marvin ; % ESPACE
X = grandma ; % ESPACE
X = randy ; % ESPACE
X = sharon ; % ESPACE
X = marvin ; % ESPACE
X = grandma ; % ESPACE
false.
```

Ceci donne tous les spécialisations de `X` telles que Sharon appartient à la même famille que `X`.
Observons certaines solutions dupliquées.

Comme l'arbre généalogique modélisé ne contient qu'une seule composante connexe, tout le monde est de la même famille.

Exercice : ajouter la familles des Broflovski, Cartman et McCormick, puis expérimenter un peu.

4.1.3. Prédicats

Pour tout $n \geqslant 1$, une *relation n-aire* entre des ensembles E_1, ..., E_n est un élément de $\mathcal{P}(E_1 \times \cdots \times E_n)$. L'*arité* de $\mathcal{R}$ est n.

Ainsi, une relation n-aire $\mathcal{R}$ entre E_1, ..., E_n est un ensemble de n-uplets $(x_1, \ldots, x_n)$ tels que $x_i \in E_i$ pour tout $1 \leqslant i \leqslant n$.

Si $(x_1, \ldots, x_n) \in \mathcal{R}$, nous disons que $\mathcal{R}$ *met en relation* x_1, ..., x_n. Cette propriété est notée par $\mathcal{R}(x_1, \ldots, x_n)$. Le fait que cette propriété est fausse est noté par $\not\mathcal{R}(x_1, \ldots, x_n)$.

Exemples

- Soit la relation $\mathcal{R}$ d'arité 3 entre $\mathbb{Z}$, $\mathbb{Z}$ et $\mathbb{Z}$ telle que $\mathcal{R}(z_1, z_2, z_3)$ si $z_1 - z_2 + z_3 = 0$.
  Par exemple, nous avons $\mathcal{R}(0, 0, 0)$ et $\mathcal{R}(4, -10, 6)$.

- Soit la relation $\mathcal{R}$ d'arité 3 entre l'ensemble des systèmes d'exploitation, l'ensemble des logiciels et l'ensemble des versions telle que $\mathcal{R}(s, l, v)$ si le logiciel l existe en version v sur le système d'exploitation s.
  Par exemple, nous avons

$$\mathcal{R}(\text{Manjaro Linux 24.1.2}, \texttt{swi-prolog}, 9.2.7), \quad \mathcal{R}(\text{Manjaro Linux 24.1.2}, \texttt{ocaml}, 5.2.0),$$
$$\mathcal{R}(\text{Manjaro Linux 24.1.2}, \texttt{i3}, 4.23).$$

Les relations n-aires sont des **généralisations des fonctions**.

En effet, une fonction $f : E_1 \times \cdots \times E_n \to E'_1 \times \cdots \times E'_{n'}$ peut se représenter la relation $\mathcal{R}_f$ d'arité $n + n'$ telle que $\mathcal{R}_f(x_1, \ldots, x_n, y_1, \ldots, y_{n'})$ si $f(x_1, \ldots, x_n) = (y_1, \ldots, y_{n'})$.

Exemples

- La fonction $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ définie par $f(z_1, z_2) := (z_2, z_1)$ est représentée par la relation $\mathcal{R}_f$ d'arité 4 vérifiant $\mathcal{R}_f(z_1, z_2, z_2, z_1)$.
- La fonction $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ définie par $+(z_1, z_2) := z_1 + z_2$ est représentée par la relation $\mathcal{R}_+$ d'arité 3 vérifiant $\mathcal{R}_+(z_1, z_2, z_1 + z_2)$.
- La fonction factorielle $! : \mathbb{N} \to \mathbb{N}$ est représentée par la relation $\mathcal{R}_!$ d'arité 2 vérifiant $\mathcal{R}_!(n, n!)$. Par exemple, $\mathcal{R}_!(4, 24)$.

Les fonctions permettent de définir des **transformations** de valeurs en entrée vers des valeurs en sortie.

Une relation n-aire permet d'établir une **propriété** que doivent vérifier des valeurs. Comme expliqué ici, ceci est plus général. En effet, dans ce contexte, un élément pourrait avoir plusieurs « images » par le biais d'une relation.

On appelle *prédicat* toute relation n-aire.

Un *système logique* est un cadre formel destiné à former des énoncés et à les démontrer.

Il est basé sur trois notions :

1. la **syntaxe**, qui explique comment former des énoncés ;
2. la **sémantique**, qui explique comment donner du sens aux énoncés ;
3. la **théorie de la démonstration**, qui explique comment démontrer qu'un énoncé est vrai.

En Prolog,

1. les énoncés sont les **requêtes** ;
2. les **définitions de prédicats** donnent du sens aux requêtes ;
3. la théorie de la démonstration est basée sur la **résolution**.

Un système logique doit vérifier deux caractéristiques importantes :

1. la **correction**, qui dit que tout ce qui est démontrable dans le système logique est vrai ;
2. la **complétude**, qui dit que tout ce qui est exprimable dans le système logique et qui est vrai y est démontrable.

La *logique du 1er ordre* est le système logique sur lequel PROLOG se base.

Une *formule* est définie récursivement comme étant

- un **terme** t ;
- la **négation** $\neg F$ d'une formule F ;
- la **disjonction** $F \vee F'$ de deux formules F et F' ;
- la **conjonction** $F \wedge F'$ de deux formules F et F' ;
- l'**implication** $F \rightarrow F'$ de deux formules F et F' ;
- la **quantification universelle** $\forall x\, F$ où x est une variable et F est une formule ;
- la **quantification existentielle** $\exists x\, F$ où x est une variable et F est une formule.

Exemple

La formule

$$\forall x \forall y \left((\mathtt{add}(x, 1, y) \wedge \mathtt{pair}(y)) \rightarrow \mathtt{impair}(y) \right)$$

est formée sur des termes utilisant les prédicats $\mathtt{pair}/1$, $\mathtt{add}/3$ et $\mathtt{impair}/1$.

PROLOG n'utilise pas cette syntaxe de la logique du 1er ordre directement.

Une **requête** `t1, ..., tn.` spécifie la formule en conjonction `t1` $\wedge \cdots \wedge$ `tn`.

Toute **variable dans une requête** est quantifiée **existentiellement**.

Exemple

En reprenant l'exemple de l'arbre généalogique, la requête `pere(marvin, X), pere(X, Y).` spécifie la formule $\exists X \, \exists Y \, \texttt{pere}(\texttt{marvin}, X) \wedge \texttt{pere}(X, Y)$

Toute **variable située dans la tête d'une clause** est quantifiée **universellement**.

Toute **variable située dans le corps d'une clause** et qui n'apparaît pas dans sa tête est quantifiée **existentiellement**.

Exemple

La clause `grand_parent(X, Y) :- parent(X, Z), parent(Z, Y).` spécifie la formule

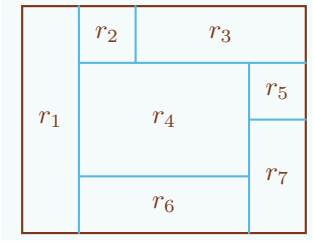
$$\forall X \, \forall Y \, ((\exists Z \, \texttt{parent}(X, Z) \wedge \texttt{parent}(Z, Y)) \rightarrow \texttt{grand\_parent}(X, Y)).$$

Nous reviendrons sur cette forme de formules logiques un peu plus loin.

4.1.4. Exemples complets

Le célèbre **théorème des quatre couleurs** annonce que toute carte peut se colorier avec quatre couleurs ou moins, sans que deux régions ayant une frontière commune ne se touchent.

Appliquons ce résultat à la carte suivante afin de trouver ses colorations autorisées :



Cette carte est constituée des sept régions r_1, ..., r_7.

Les **contraintes** de coloration font que si r_i et r_j sont adjacentes, alors la couleur de r_i doit être différente de la couleur de r_j.

Appelons `c1`, `c2`, `c3` et `c4` nos quatre couleurs.

Nous commençons par définir le prédicat `couleur/1` tel que `couleur(C)` si `C` est une couleur. Il se définit par

```
couleur(c1).
couleur(c2).
couleur(c3).
couleur(c4).
```

Exemples

Bien entendu, nous avons

```
?- couleur(c2).
true.

?- couleur(c3).
true.

?- couleur(miaou).
false.
```

L'intérêt principal réside dans le fait que le prédicat `couleur/1` est un **générateur** des couleurs autorisées.

Exemple

```
?- couleur(C).
C = c1 ;
C = c2 ;
C = c3 ;
C = c4.
```

Soit `couleurs_compatibles/2` le prédicat tel que `couleurs_compatibles(C1, C2)` si `C1` et `C2` sont deux couleurs compatibles (c'est-à-dire, différentes). Il se définit par

```
couleurs_compatibles(C1, C2) :-
    couleur(C1),
    couleur(C2),
    C1 \= C2.
```

Exemples

Voici quelques utilisations directes :

```
?- couleurs_compatibles(c1, c2).
true.

?- couleurs_compatibles(c1, c1).
false.

?- couleurs_compatibles(c2, miaou).
false.
```

Exemple

Voici une utilisation comme générateur :

```
?- couleurs_compatibles(C1, C2).
C1 = c1, C2 = c2 ;
C1 = c1, C2 = c3 ;
C1 = c1, C2 = c4 ;
C1 = c2, C2 = c1 ;
C1 = c2, C2 = c3 ;
C1 = c2, C2 = c4 ;
C1 = c3, C2 = c1 ;
C1 = c3, C2 = c2 ;
C1 = c3, C2 = c4 ;
C1 = c4, C2 = c1 ;
C1 = c4, C2 = c2 ;
C1 = c4, C2 = c3 ;
false.
```

Soit `coloration/7` le prédicat tel que `coloration(C1, C2, C3, C4, C5, C6, C7)` si la coloration de la carte initiale obtenu en colorant r_1 de la couleur `C1`, r_2 de la couleur `C2`, ..., r_7 de la couleur `C7` est compatible. Il se définit par

```
coloration(C1, C2, C3, C4, C5, C6, C7) :-
    couleurs_compatibles(C1, C2),
    couleurs_compatibles(C1, C4),
    couleurs_compatibles(C1, C6),
    couleurs_compatibles(C2, C3),
    couleurs_compatibles(C2, C4),
    couleurs_compatibles(C3, C4),
    couleurs_compatibles(C3, C5),
    couleurs_compatibles(C4, C5),
    couleurs_compatibles(C4, C6),
    couleurs_compatibles(C4, C7),
    couleurs_compatibles(C5, C7),
    couleurs_compatibles(C6, C7).
```

Ce prédicat s'obtient en considérant toutes les régions r_i et r_j de la carte qui ont une frontière commune et en ajoutant la condition que les couleurs `Ci` et `Cj` doivent être compatibles. Cela se fait, avec les définitions précédentes, par `couleurs_compatibles(Ci, Cj)`.

Les solutions sont

```
?- coloration(C1, C2, C3, C4, C5, C6, C7).
C1 = C3, C3 = C7, C7 = c1, C2 = C5, C5 = C6, C6 = c2, C4 = c3 ;
C1 = C3, C3 = c1, C2 = C5, C5 = C6, C6 = c2, C4 = c3, C7 = c4 ;
C1 = C3, C3 = C7, C7 = c1, C2 = C6, C6 = c2, C4 = c3, C5 = c4 ;
C1 = C5, C5 = c1, C2 = C6, C6 = c2, C3 = C7, C7 = c4, C4 = c3 ;
...
C1 = C3, C3 = c4, C2 = C5, C5 = C6, C6 = c3, C4 = c2, C7 = c1 ;
C1 = C3, C3 = C7, C7 = c4, C2 = C5, C5 = C6, C6 = c3, C4 = c2 ;
false.
```

Il y en a beaucoup.

Exercice : expérimenter avec d'autres cartes (en mettant à jour `coloration/7` de sorte à tenir compte des nouvelles frontières).

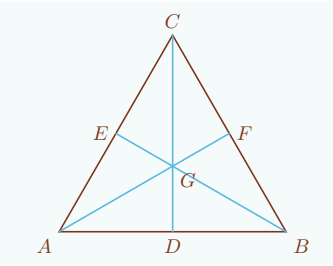
Nous verrons plus tard comment **collecter dans une liste** toutes les solutions calculées. Ceci nous sera utile pour, par exemple dénombrer les solutions ou encore pour réaliser un traitement supplémentaire sur ces dernières.

Intéressons-nous à un problème de dénombrement concret.

Voici une figure géométrique.

Nous souhaitons **dénombrer les triangles** qui la composent.

Par exemple, ABC, ABG et BCE sont de tels triangles.



Notre 1re tâche consiste à représenter cette figure. Observons que les coordonnées exactes des points n'importent pas pour notre objectif. Seule la **structure de la figure intervient**.

Nous allons, pour la représenter, utiliser

- un atome par point : `a`, `b`, `c`, `d`, `e`, `f` et `g` ;
- un atome par segment maximal : `ab`, `ac`, `af`, `bc`, `be` et `cd` ;
- un prédicat qui teste si un segment contient un point.

Soit `segment_point/2` le prédicat tel que `segment_point(SEG, P)` si le segment `SEG` contient le point `P`. Il se définit par

```
segment_point(ab, a).
segment_point(ab, b).
segment_point(ab, d).
segment_point(ac, a).
segment_point(ac, c).
segment_point(ac, e).
segment_point(af, a).
segment_point(af, f).
segment_point(af, g).
segment_point(bc, b).
segment_point(bc, c).
segment_point(bc, f).
segment_point(be, b).
segment_point(be, e).
segment_point(be, g).
segment_point(cd, c).
segment_point(cd, d).
segment_point(cd, g).
```

Cet ensemble de faits représente notre figure « en dur ».

Ce prédicat n'est pas générique. Pour adapter notre solution à d'autres figures, il faudra mettre à jour ce prédicat.

Nous allons mettre au point des prédicats `triangle_vN/3` qui sont tels que `triangle_vN(P1, P2, P3)` si les trois points `P1`, `P2` et `P3` forment un triangle dans la figure.

La requête `triangle_vN(P1, P2, P3).` va nous permettre d'**engendrer toutes les solutions**, qu'il nous suffira de compter (comme mentionné avant, il existe des méthodes automatiques pour compter les solutions, chose qui sera vue par la suite).

Nous donnerons plusieurs versions que nous affinerons progressivement,

Voici une première tentative :

```
triangle_v1(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3).
```

Ceci est très naturel : nous affirmons que pour avoir un triangle formé par les points `P1`, `P2` et `P3`, nous devons avoir trois segments `P1_P2`, `P2_P3` et `P1_P3` contenant les points de manière cohérente.

Malheureusement,

```
?- triangle_v1(P1, P2, P3).
P1 = P2, P2 = P3, P3 = a ;
...
```

construit une solution formée d'un **triangle réduit à un point**.